Solving Secular Equations stably and Efficiently

Dimitri Bouche, Gauthier Schweitzer

1 Problem presentation

Our objective is to solve the following secular equation :

$$f(x) = \rho + \sum_{i=1}^{n} \frac{\zeta_i^2}{\delta_i - x} \tag{1}$$

Its roots correspond the eigenvalues of a diagonal matrix $D = diag(\delta_1, ...\delta_n)$ with a rank-1 perturbation :

$$D + \frac{zz^T}{\rho}$$

We set $z = (\zeta_1, ...\zeta_n)$. Without loss of generality, we assume that $(\delta_1, ...\delta_n)$ is such that

$$\delta_1 < \delta_2 < ... < \delta_n$$

and every $\zeta_i \neq 0$.

f has n roots, $n-1$ are in $[\delta_k, \delta_{k+1}]$ while one is in $[\delta_n, +\infty]$ (if $\rho > 0$ and we can assume it without loss of generality). The purpose of our work is to find numerically these roots by using Gragg's aglorithm. All of our formulas are derived from [1]. We will first present the algorithm and the main formulas that we have been using. After that, we will discuss its implementation using CUDA. Finally, we will show some results comparing the performance of the following procedure on CPU or on GPU.

2 Algorithm

2.1 Iteration Formulas

2.1.1 Interior roots

En se plaçant à un y donné, cherche à résoudre une équation du second degré ayant pour inconnu η afin que $y+\eta$ soit plus proche de la racine que l'on cherche que y. En posant :

$$\Delta_k = \delta_k - y$$

$$\Delta_{k+1} = \delta_{k+1} - y$$
$$a = (\Delta_k + \Delta_{k+1})f(y) - \Delta_k\Delta_{k+1}f'(y)$$
$$b = \Delta_k\Delta_{k+1}f(y)$$

η est donné par :

$$\eta = \frac{a - \sqrt{a^2 - 4bc}}{2c} \quad si \quad a \leq 0 \tag{2}$$
$$\eta = \frac{2b}{a + \sqrt{a^2 - 4bc}} \quad si \quad a > 0 \tag{3}$$

Tiré de la section 3.3 page 15 :

c est un degré de liberté qui nous est donné pour l'interpolation. Dans le cas de l'algorithme de Gragg, c est choisi tel que l'interpolation rationnelle de la fonction séculaire en y coïncide aussi avec la dérivée seconde de cette dernière en ce même point. La formule pour c est alors :

$$c = f(y) - (\Delta_k + \Delta_{k+1})f'(y) + \Delta_k\Delta_{k+1}\frac{f''(y)}{2} \tag{4}$$

En mettant ensemble (2) et (4), on peut donc calculer η, reste à incrémenter y en lui ajoutant η. C'est la formule d'itération de Gragg pour les racines intérieures.

2.1.2 Exterior root ($k = n$)

Il faut adapter un peu la précédente procédure. On garde la même interpolation que pour $k = n - 1$ donc ;

$$\Delta_{n-1} = \delta_{n-1} - y$$
$$\Delta_n = \delta_n - y$$
$$a = (\Delta_{n-1} + \Delta_n)f(y) - \Delta_{n-1}\Delta_n f'(y)$$
$$b = \Delta_{n-1}\Delta_n f(y)$$
$$c = f(y) - (\Delta_{n-1} + \Delta_n)f'(y) + \Delta_{n-1}\Delta_n\frac{f''(y)}{2}$$

Sauf que η est donnée par une version modifiée de (2) :

$$\eta = \frac{a + \sqrt{a^2 - 4bc}}{2c} \quad si \quad a \geq 0 \tag{5}$$
$$\eta = \frac{2b}{a - \sqrt{a^2 - 4bc}} \quad si \quad a < 0 \tag{6}$$

On peut là aussi calculer η et ainsi l'ajouter à y pour se rapprocher de la racine.

2.2 Initilization

2.2.1 Educated guess

In the article, the authors suggest initial values and give theoretical reasons why they should lead to a nice convergence rate. Choix des valeurs initiales : section 4 - Initial guesses pages 18 - 19 - 20

Interior roots Pour les racines intérieures : voir fin de la page 19 et formules (42), (43) et (44).

Exterior root ($k = n$) Pour la racine extérieure voir la page 21 (notamment partie avec la distinction de cas et les formules (46) et (47)).

2.2.2 Random guess

We also also tried a method consisting in randomly picking a number between the two poles. On the exterior part, the upper bound of the space on which we draw randomly is the same as the one given in the paper.

3 Implementation in C

We have developped two main algorithms, one using the CPU and one using the GPU

3.1 CPU

We won't go very deep into details for this implementation. One should however mention that we have built two sub-algorithms using CPU. The first one uses *float* variables while the second one has double precision and uses *double*. We will see in the results part that these specificities have strong implications, both for the running time and for the precision of the estimated roots.

3.2 GPU

Why using a GPU Secular equations solving is a task that has two features that make it very interesting for GPU computing :

— it requires important computational power ;

— it is by essence highly parallelizable.

The second point is more interesting to discuss. To solve secular questions, one has to solve n different problems, each one corresponding to the computation of one root. All of these subproblems are independent since one does not need the root on one interval to compute a root on another one. Therefore, it appears natural to parallelize these computations. Actually, parallelization goes even further.

__global__ functions We have three distinct kernels, each of them corresponding to a task to parallelize. These kernels are launched one after another :

1. The first kernel *square_kernel* computes the square of each ζ_i and the squared norm of the ζ vector. It uses a standard grid dimension $<< 1024, 512 >>$ to take full profit of the NVIDIA 1080 GPU. Therefore, each thread is computing the square of one coordinate and is doing an AtomicAdd (to prevent concurrent writing) to obtain the square ;
2. The second kernel *initialize_x0_kernel* is used to initialize the root-finding process on each of the intervals. Therefore, each thread has a given interval on which it performs calculations to set the initial value of λ. Once more, the grid is $<< 1024, 512 >>$;
3. Finally, the last kernel is the main one *find_roots_kernel*. Starting from the different inital values, it performs Gragg's algorithm to obtain the roots. As explained, each thread computes one root. Grid size is $<< 1024, 512 >>$.

Other functions The rest of the functions that we are using are either standard host functions (called from the host to be performed on the host), or *__device__* ones (called from the device to be performed on the device).

Access to memory For many workloads, the performance benefits of parallelization are hindered by the large and often unpredictable overheads of launching GPU kernels and of transferring data between CPU and GPU. Therefore, we tried to limit as much as possible communications between the host and the device. We mainly used registers memory, that is the fastest one. These variables stored in registers are local. We have also tried a version of the algorithm using shared memory for values that are accessed multiple times.

4 Results

4.1 The scripts that we have used

4.1.1 Testing the different algorithms individually

— main_cpu_double.cu : CPU is used, with a double precision to compute the n roots of a secular equation, n being given by the user.
— main_cpu_float.cu : CPU is used, with a single precision to compute the n roots of a secular equation, n being given by the user
— main_gpu.cu : GPU is used, with a single precision to compute the n roots of a secular equation, n being given by the user.

4.1.2 Comparing the performance

Directly in the terminal

— comp_console.cu : Both CPU and GPU with single precision are used to compute the n roots of a secular equation, n being given by the user. The differential in performance can be seen immediately in the console (running time and magnitude of the loss)

Generating a csv that can be graphically read in a Python notebook

- comp_table.cu : Both CPU and GPU with single precision are used to compute the n roots of a secular equation. A range of n is given by the user and performance (running time and magnitude of the error) is stored on a csv file ('result.csv'). To compare them on a fair basis, the user can choose to run the test several time. For each n being tested, the GPU is warmed-up before the first iteration. To try with high values of n, the user can also choose not to compute the roots with the CPU (only GPU)
- double.cu : Performs the same task with double precision for the CPU (output is 'result_double.csv')
- memory.cu : Performs the same task with the GPU, in a version using shared memory (output is 'result_mem.csv')
- initialization.cu : Performs the same task with CPU, the algorithm being initialized randomly as described before (output is 'result_init.csv')

Cf notebook "Charts" à commenter, qui regroupe les graphiques de performance sur

Références

[1] Ren-Cang Li. "Solving secular equations stably and efficiently". In : (1993).